SDK 导入

cocos2dx 配置

在使用 SDK 提供的方法之前，需要先初始化 SDK 环境，调用：

BmobSDKInit::initialize(“your app_id”,”your app_key”);

your app_id:控制台申请的 Application ID;

your app_key：控制台申请的 REST API Key;

应用程序

在 Bmob 平台注册后，每个账户可创建多个应用程序，创建的每个应用程序有各自的 Application ID，应用程序将凭其 Application ID 使用 BmobSDK。

应用安全

请大家在使用 Bmob 开发应用程序之前，仔细阅读“数据与安全”的文档：
http://docs.bmob.cn/datasafety/index.html?menukey=otherdoc&key=datasafety

数据类型

目前为止，我们支持的数据类型
CCString、CCInteger、CCBool、CCArray、CCDictionary、map、string 以及 BmobObject 对象类型。

对象

一个数据对象（BmobObject 类和子类，其中 BmobObject 继承 Cocos2dx 中的 CCObject 类）对应于 Bmob 后台的一个数据表。

数据对象

Bmob 存储的数据是建立在 BmobObject 基础上的，所以任何要保存的数据对象必须继承自 BmobObject 类。BmobObject 类本身包含 objectId、createdAt、updatedAt、ACL 四个默认的属性，objectId 是数据的唯一标示，相当于表的主键，createdAt 是数据的创建时间，updatedAt 是数据的最后修改时间，ACL 是数据的操作权限。

如游戏中使用 GameScore 表来记录玩家的比分信息，其中表的字段有：score（分数）、playerName（玩家名字）、info（玩家头像）属性，这个数据对象如下定义：

//必须要继承自 BmobObject 类

```
class GameScore:public BmobObject{
public:
    GameScore(string tableName);
    ~GameScore();
private:
    CC_SYNTHESIZE(std::string, m_playerName, PlayerName);
    CC_SYNTHESIZE(int,m_score,Score);
    CC_SYNTHESIZE(std::string,m_info,Info);
};
```

注：对于开发者来说，不需要对 objectId、createdAt、updatedAt、ACL 四个属性进行定义，已经在 BmobObject 类中默认定义了。

类名和表名的关系

- Bmob 官方推荐类名和表名完全一致的映射使用方式， 即如，上面的 GameScore 类，在使用 SDK 的接口保存数据的时候，传递对应的类名，在后台创建的表名就和传递名一样。

- 如果不想创建和类名一样的表名，传递其他的名字即可。

添加数据

添加数据使用 BmobObject 类的提供的 save 方法并传递监听接口指针，将当前对象的内容

保存到 Bmob 后台服务。

方法原型：

```
    virtual void save(BmobSaveDelegate* delegate);
```

例如，将玩家为 Habrrier 的信息保存到 GameScore 数据表中，代码如下：

```
        GameScore* game = new GameScore("GameScore");
        game->autorelease();
        game->setName("Habrrier");
        game->setScore(670);
        game->setInfo("My name is Habrrier");
        game->clear();
        game→enParamsToHttp("playerName",CCString::createWithFormat("%s",game→
                        game->getName().c_str()));
        game->enParamsToHttp("score",CCInteger::create(game->getScore()));
        game->enParamsToHttp("info",CCString::createWithFormat("%s",game-
>getInfo().c_str()));
        game->save(this);
```

上面调用 enParamsToHttp 方法是将上面的属性转换成 key-value 放入 http 请求中，只有调用该方法，才能使用 save 保存数据到后台。

同时 this 代表的类中实现 BmobSaveDelegate 接口，监听保存数据的返回状态，回调方法是：

```
void onSaveSucess(const void* data){
    //数据添加成功
}
void onSaveError(int code,const void* data){
    //数据添加失败
}
```

如果对返回状态不关心可以传递 NULL 作为 save 的参数。
实现 BmobSaveDelegate 接口如下：

```
class HelloWorld:public BmobSaveDelegate{
…
public:
    virtual void onSaveSucess(const void* data){//To DO}
    virtual void onSaveError(int code,const void* data){//To Do}
….
};
```

运行以上代码，如果添加成功，你可以在 Bmob 提供的后台的数据浏览中看到类似这样的结果：

createdAt:"2015-10-26 11:34:45",info:" info 2015-10-25 12:22:12",name:"Lingo",objectId:"422185deb7",score:12,updatedAt:"2015-10-26 11:34:45"

这里需要注意的是：

1. 如果服务器端不存在 GameScore 表，系统将自动建表该表，并插入数据。
2. 如果服务器端已经存在 GameScore 表，那就会将该条数据保存到对应的表中。
3. 每个 BmobObject 对象都有几个默认的键(数据列)是不需要开发者指定的，objectId 是每个保存成功数据的唯一标识符。createdAt 和 updatedAt 代表每个对象(每条数据)在服务器上创建和最后修改的时间。这些键(数据列)的创建和数据内容是由服务器端自主生成。因此，使用 save 方法时，不需要调用 setObjectId 方法，否则会出现提示："It is a reserved field: objectId(105)" --表明 objectId 为系统保留字段，不允许修改。

修改数据

修改数据主要是调用 SDK 重的 update 方法，将对象的 objectId 和实现的监听接口传递给该方法，以指明要更新的数据。方法原型：

virtual void update(string objectId,BmobUpdateDelegate* delegate);

例如：将 GameScore 表中 objectId 为 2e0f067922 的游戏分数修改为 1000.

```
GameScore* game = new GameScore("GameScore");
game->autorelease();
game->setScore(1000);
game->clear();
game->enParamsToHttp("score",CCInteger::create(game->getScore()));
game->update("2e0f067922",this);
```

其中 this 必须实现 BmobUpdateDelegate 接口，以监听更新状态。

```
void onUpdateSucess(void* data){
    //数据更新成功
}
void onUpdateError(int code,void* data){
```

```
        //数据更新失败
    }
```

同样的和保存数据一样，使用 set 方法来更新数据也需要调用 enParamsToHttp 将数据放入 http 中。

SDK 提供了另一种方法来更新数据，通过调用 BmobobJect 类中的 setValue（key，value）方法，只需要传入 key 和要更新的值，如：

```
        GameScore* game = new GameScore("abcdef");
        game->autorelease();

        game->clear();
        game->setValue("score",CCFloat::create(1000));
        game->update("2e0f067922",this);
```

更新数据时，如果对更新状态需要监听，需实现 BmobUpdateDelegate 接口，并传递给 update 方法。

接口 BmobUpdateDelegate 实现如下：

```
class HelloWorld:public BmobUpdateDelegate{
…
public:
    virtual void onUpdateSucess(const void* data){//To Do}
    virtual void onUpdateError(int code const void* data){//To Do}
..
};
```

删除数据

从服务器删除对象使用 BmobObject 对象的 del 方法（注：在这里不用 delete 方法是因为在 cocos2dx 重，delete 是系统方法）并传递一个实现的监听接口参数，方法原型：

```
    virtual void del(BmobDeleteDelegate* delegate);
```

例如：将 GameScore 表中 objectId 为 0875c8a278 的数据删除。

```
        GameScore* game = new GameScore("abcdef");
        game->autorelease();
        game->setObjectId("0875c8a278");
        game->del(this);
```

在这里 this 类必须实现 BmobDeleteDelegate 接口，以便监听状态，如果传递 NULL，说明删除状态不监听。其监听回调方法是：

```
    void onDeleteSucess(void* data){
        //删除数据成功
    }
    void onDeleteErrpr(int code,void* data){
        //删除数据失败
```

```
    }
```

接口 BmobDeletedelegate 实现如下：

```
class HelloWorld:public BmobDeleteDelegate{
..
public:
    virtual void onDeleteSucess(const void* data){//To Do}
    virtual void onDeleteError(int code,const void* data){//To Do}
..
};
```

删除数据除了使用上面调用 setObjectId 来设置 objectId 更新以外，还可以调用 del 的一个重载方法：

```
    virtual void del(string objectId,BmobDeleteDelegate* delegate);
```

来更新，直接通过传递 objectId 实现更新。如：

```
        GameScore* game = new GameScore("GameScore");
        game->autorelease();
        game->del("0875c8a278",this);
```

注意：删除数据只能通过 objectId 来删除，目前不提供查询条件方式的删除方法。

删除字段的值

可以在一个对象中删除一个字段的值，通过 BmobObject 的 remove 结合 update 操作，使用 remove 方法设置要删除的字段，使用 update 方法实现删除。如将 GameScore 表重 objectId 为 0875c8a278 的 info 字段的值删除：

```
    GameScore* game = new GameScore("GameScore");
    game->autorelease();
    game->setObjectId("0875c8a278");
    game->remove("info");
    game->update(this);
```

在这里 this 类必须实现 BmobUpdateDelegate 接口，以便监听状态，如传递 NULL，说明对状态不进行监听。其监听回调方法是：

```
    void onUpdateSucess(void* data){
        //删除数据成功
    }
    void onUpdateErrpr(int code,void* data){
        //删除数据失败
    }
```

接口 BmobUpdateDelegate 的实现如下：

```
class HelloWorld:public BmobUpdateDelegate{
….
public:
    virtual void onUpdateSucess(const void* data){ //To Do}
    virtual void onUpdateError(int code,const void* data){//To Do}
```

```
};
```

查询数据

数据的查询可能是每个应用都会频繁使用，BmobSDK 中提供了 BmobQuery 类，提供了多种方法实现不同条件的查询。

注：在做新的查询之前一定要调用 BmobQuery 的 clear 方法，否则结果不可预知。

查询所有数据

查询某个数据表中的所有数据使用 BmobQuery 提供的 findObjects 方法，参数是监听接口。方法原型：

```
virtual void findObjects(BmobFindDelegate* delegate);
```

例如：查询 GameScore 表中的所有数据：

```
BmobQuery* query = new BmobQuery("GameScore");
query->autorelease();
query->findObjects(this);
```

注：如果需要监听查找状态以及查询结果，必须实现 BmobFindDelegate 接口，上面的 this 必须实现该接口。接口中的回调方法方法：

```
    virtual void onFindSucess(const void* data) {
        //查询数据成功回调，data 是查询返回的数据
    }
    virtual void onFindError(int code,const void* data) {
        //查询数据失败回调，data 是失败信息
    }
class HelloWorld:public BmobFindDelegate{
…
public:
```

```
    virtual void onFindSucess(const void* data){//To Do}
    virtual void onFindError(int code,const void* data){//To Do}
  …..
};
```

同样的也可以使用 findObjects 的一个变体，通过直接指定表明查询，方法原型是：

```
    virtual void findObjects(string tableName,BmobFindDelegate* delegate) = 0;
```

如果对查询结果和状态不关系的，可以传递 NULL 参数。

同时也可以通过设置 where 条件来查询满足条件的所有数据。

如可以使用 BmobQuery 中的 findObjects 方法查询 GameScore 表中 playerName 爲 xiaoming 的所有数据：

```
        BmobQuery* query = new BmobQuery("GameScore");
        query->autorelease();
        string name = "xiaoming";
        query->clear();
        query->addWhereEqualTo("playerName",CCString::createWithFormat("%s",
                                name.c_str()));
        query->findObjects(this);
```

注：在不需要 where 查询或者是查询全部数据之前，先调用 clear 清除之前设置过的 where 条件，或者是当重新设置 where 条件，那之前必须先调用 clear 函数以清除之前的 where 条件。

查询单条数据

同样的可以根据 objectId 直接获取单条数据对象，查询调用 BmobQuery 提供的 getObject 方法并传递 objectId 以及回调接口查询。方法原型：

```
    virtual void getObject(string objectId,BmobGetDelegate* delegate);
```

例如：在 GameScore 表中查询 objectId 为 c660c4166d 的人员信息。

```
BmobQuery* query = new BmobQuery("GameScore");
 query->autorelease();
 query->getObject("c660c4166d",this);
```

注：如果需要监听查询的状态以及查询结果，必须实现 BmobGetDelegate 接口，上面的 this 实现了该接口。接口中的回调方法：

```
virtual void onGetSucess(const void* data) {

    //查询单条数据成功，data 存储了查询的结果

}

virtual void onGetError(int code,const void* data) {

    //查询单条数据失败，code 返回状态码，data 存储失败信息

}
```

接口 BmobGetDelegate 的实现如下：

```
class HelloWorld:public BmobGetDelegate{
…
public:
    virtual void onGetSucess(const void* data){//To Do}
    virtual void onGetError(int code,const void* data){//To Do }
    …..
};
```

限制查询

查询数据可以使用 BmobQuery 提供的 setLimit 和 setSkip 方法来做一些限制。限制结果查询返回数量，使用 setLimit 方法；限制结果跳转的页数，使用 setSkip 方法。方法原型是：

```
virtual void setLimit(int limit);
virtual void setSkip(int skip);
```

如限制跳转到结果的第 10 页，并且仅查询 20 条数据的使用方法是：

```
BmobQuery* query = new BmobQuery("GameScore");
 query->autorelease();

query->setLimit(20);
query->setSkip(10);

query->findObjects(this);
```

需要监听查询状态和结果，需要实现 BmobFindDelegate 接口，上面的 this 实现了该接口。

注：如果 setSkip 设置的值超过了结果的页数，那返回将没有结果数据。

查询条件

在查询的使用过程中，基于不同条件的查询是非常常见，BmobQuery 同样也支持不同条件的查询。

比较查询

要查询特定键的值，可以使用 BmobQuery 提供的 addWhereEqualTo 方法指定条件。如要过滤掉特定键的值可以使用 addWhereNotEqualTo 方法指定条件，之后调用 findObjects 方法触发查询。方法原型是：

```
virtual void addWhereEqualTo(string seg,CCObject *object);

virtual void addWhereNotEqualTo(string seg,CCObject *object);
```

比如需要查询 name 等于“Barbie”的数据时可以这样写：

```
query->addWhereEqualTo("name", "Barbie");

query→findObjects(this);
```

同样可以在查询操作中添加多个约束条件，来查询符合的数据。

如查询 GameScore 表中 playerName 不等于 Bridder，且 score 等于 10989 的数据：

```
BmobQuery* query = new BmobQuery("GameScore");
query->autorelease();
string name = "Bridder";
query->clear();
query->addWhereEqualTo("score",CCFloat::create(10989));
query->addWhereNotEqualTo("playerName",CCString::createWithFormat("%s",
                    name.c_str()));
query->findObjects(this);
```

以下还包含了各种不同条件的比较查询：

小于条件：

```
virtual void addWhereLessThan(string seg,CCObject* object) ;

//添加玩家分数小于 2345 的条件//

query->addWhereLessThan("score",CCInteger::create(2345));
```

小于等于条件：

```
virtual void addWhereLessThanOrEqualTo(string seg,CCObject* object);

//添加玩家分数小于等于 2345 的条件//

    query->addWhereLessThanOrEqualTo("score",CCInteger::create(2345));
```

大于条件：

```
    virtual void addWhereGreaterThan(string seg,CCObject* object);

    //添加玩家分数大于 2334 的条件//

    query->addWhereGreaterThan("score",CCInteger::create(2334));
```

大于等于：

```
    virtual void addWhereGreaterThanOrEqualTo(string seg,CCObject*
                                              object) ;
    //添加玩家分数大于等于 2345 的条件
    query->addWhereGreaterThanOrEqualTo("score",CCInteger::create(2345));
```

注：不能添加具有矛盾的查询条件，否则将得不到想要的结果！

子查询

如果需要查询匹配几个不同值的数据，可以使用 BmobQuery 提供的 addWhereContainedIn 方法，方法原型：

```
    virtual void addWhereContainedIn(string seg,CCObject* array) ;
```

如：要查询 GameScore 表中 playerName 为“Barbie”，“Joe”，“Julia”三个人的成绩：

```
    string[] names = {"Barbie", "Joe", "Julia"};
    CCArray* array = CCArray::create();
    array->addObject(CCString::createWithFormat("%s",name[0].c_str()));
    array->addObject(CCString::createWithFormat("%s",name[1].c_str()));
    array->addObject(CCString::createWithFormat("%s",name[2].c_str()));
    query->addWhereContainedIn("playerName", array);
```

相反，如果想查询排除“Barbie”，“Joe”，“Julia”这三个人的其他同学的信息，你可以使用 addWhereNotContainedIn 方法来实现。方法原型是：

```
    virtual void addWhereNotContainedIn(string seg,CCObject* array) ;


    query->addWhereNotContainedIn("playerName", array);
```

之后调用 findObjects 方法查询：

```
    query->findObjects(this);
```

排序

对应数据的排序，如数字或字符串，可以使用升序或降序的方式来控制查询数据的结果顺序，需要进行排序查询，可以使用 BmobQuery 提供的 order 方法，其方法原型：

```
virtual void order(string key);
```

如对 GameScore 重的 score 段进行升序或降序查询：

```
query->order("score");//升序//
query->order("-score");//降序//
query->order("-score,PlayerName");//对多个字段进行排序查询//
```

说明：多个字段排序时，先按第一个字段进行排序，再按第二个字段进行排序，依次进行。

数组查询

对于字段类型为数组的情况，需要查找字段中的数组值包含有 xxx 的对象，可以使用 addWhereContainsAll 方法，方法原型是：

```
virtual void addWhereContainsAll(string seg,CCArray* array) ;
```

如查询有 Read、Write、Coffee 的人：

```
CCArray* array1 = CCArray::create();
string name1[] = {"Read","Write","Coffee"};
array->addObject(CCString::createWithFormat("%s",name1[0].c_str()));
array->addObject(CCString::createWithFormat("%s",name1[1].c_str()));
array->addObject(CCString::createWithFormat("%s",name1[2].c_str()));

query->addWhereContainsAll("hobby", array);

query->findObjects(this);
```

列值是否存在

如果想查询某个列是否存在，可以使用 addWhereExists 方法，方法原型是：

```
virtual void addWhereExists(string column);

//查询 username 有值的数据
query->addWhereExists("username");
```

如果想查询某个列值不存在，则可以用 addWhereDoesNotExists 方法，方法原型是：

```
virtual void addWhereDoesNotExists(string column);

//查询 username 字段没有值的数据
query->addWhereDoesNotExists("username");
```

```
query->findObjects(this);
```

查询指定列

可以限定查询返回的字段，通过 BmobQuery 的 addQueryKey 方法添加 where 条件。方法原型是：

virtual void addQueryKeys(string column) ;

通过传入 keys 参数，值为用一个逗号分隔的字段名称列表，为了获取对象只包含 score 和 playerName 字段(还有特殊的内置字段比如 objectId,createdAt 和 updatedAt)，请求如下：

query->addQueryKeys(”score,playerName“);

之后调用 findObjects 或者是 getObject 来查询。

查询个数

统计满足查询条件的对象数量，且不需要获取所有匹配对象的具体数据信息，可直接使用 count 替代 findObjects。例如，查询 playerName 为 Barrier 玩家玩的游戏场数：

```
BmobQuery* query = new BmobQuery("GameScore");
query->autorelease();
string name = "Barrier";
query->addWhereEqualTo("playerName",CCString::createWithFormat("%s",
                    name.c_str()));
query->count(this)
```

如果需要获得返回的数据加数量，可以使用 BmobQuery 的 count 重载方法，count(BmobCountDelegate* delegate,bool sign)并传递一个 bool 值，false 是不返回数据（和直接调用 count 一样的结果），true 返回数据。如：

```
BmobQuery* query = new BmobQuery("GameScore");
query->autorelease();
string name = "lingo";
query→addWhereEqualTo("playerName",CCString::createWithFormat("%s",
                    name.c_str()));
query->count(this,true);
```

要获得查询的结果，需要实现 BmobCountDelegate 接口，并实现其中的回调方法：

```
virtual void onCountSucess(const void* data){

    //返回查询数据
    //数据格式：{"count":3,"results":[]}
}
```

```
virtual void onCountError(int code,const void* data){
    //返回失败信息与数据
}
```

数组

对于数组类型数据，BmobSDK 提供了 3 种操作来原子性地修改一个数组字段：

add 在一个数组字段的后面添加一些指定的对象(包装在一个数组内)

setValue 在一个数组字段里面修改其中的值

removeAll 从一个数组字段的值内移除指定的数组中的所有对象

添加数组数据

可以使用 BmobObject 中的 add 方法实现数组的添加，函数原型：

```
/**
*  只添加一个数据
*/
virtual void add(string column,CCObject* object);
/**
* 同时添加多个数据
*/
virtual void add(string column,CCArray* array);
```

之后调用 BmobObject 提供的 save 方法来添加到服务，如在 GameScore 表中添加一个数组名为 list 的数组，其中只有一个数据：

```
GameScore* game = new GameScore("GameScore");
game->autorelease();
game->add("list",CCInteger::create(234));
game->save(this);
```

如果要一次添加含多个数据的数组到表中，同样的调用 add 方法，只是需要传递包含多个元素的数组：

```
GameScore* game = new GameScore("GameScore");
game->autorelease();
CCArray* array = CCArray::create();
array->addObject(CCInteger::create(120));
```

```
array->addObject(CCInteger::create(234));
game->add(“list”,array);
game->save(this);
```

更新数组

更新数组使用 BmobObject 提供的 setValue 方法来设置需要更新的数组元素，其方法原型是：

```
void setValue(string key,cocos2d::CCArray* array);
```

更新需要传递一个数组作为参数，如需要更新 GameScore 表中 objectId 为"5ec74b2297“list 数组的数据：

```
GameScore* game = new GameScore("GameScore");
game->autorelease();
CCArray* array = CCArray::create();
array->addObject(CCInteger::create(123));
array->addObject(CCInteger::create(234));
game->setValue("list",array);
game->update(“5ec74b2297”,this);
```

查询数组

对于字段类型为数组的情况，可以查找字段中的数组值包含有 xxx 的对象，查询使用 BmobQuery 提供的 addWhereContainsAll 方法来查询，方法原型是：

```
void BmobQuery::addWhereContainsAll(string seg,CCArray* array)；
```

如查询 GameScore 表中含有 hobby 数组字段且包含阅读、唱歌的数据：

```
BmobQuery* query = new BmobQuery("GameScore");
query->autorelease();
CCArray* array = CCArray::create();
string name1[] = {"阅读","唱歌"};
array->addObject(CCString::createWithFormat("%s",name1[0].c_str()));
 array->addObject(CCString::createWithFormat("%s",name1[1].c_str()));
query->addWhereContainsAll("name",array);
query->findObjects(this);
```

删除数组数据

删除数组使用 BmobObject 提供的 removeAll 来设置要删除的数组字段，其函数原型是：

```
virtual void removeAll(string name,CCArray* array);
```

如：删除 GameScore 数据表中 objectId 为“5ec74b2297”的 含有 list 数组，且其中含有 123 和 234 的值：

```
GameScore* game = new GameScore("GameScore");
game->autorelease();
game->setObjectId("");
CCArray* array = CCArray::create();
array->addObject(CCInteger::create(123));
array->addObject(CCInteger::create(234));
game->removeAll("list",array);
game->update(this);
```

统计相关的查询

Bmob 的统计查询，提供以下关键字或其组合的查询操作：

Key	Operation
groupby	分组操作
groupcount	返回每个分组的总记录
sum	计算总和
average	计算平均值
max	计算最大值
min	计算最小值
having	分组中的过滤条件

为避免和用户创建的列名称冲突，Bmob 约定以上统计关键字（sum，max，min)的查询结果值都用 '_(关键字)+首字母大写的列名' 的格式，如计算玩家得分列名称为 score 总和的操作，则返回的结果集会有一个列名为_sumScore。average 返回的列为 '_avg+首字母大写的列名'，有 groupcount 的情形下则返回_count。

以上关键字除了 groupcount 是传 Boolean 值 true 或 false，having 传的是和 where 类似的 json 字符串，但 having 只应该用于过滤分组查询得到的结果集，即 having 只应该包含结果集中的列名如{"_sumScore":{"$gt":100}}，其他关键字必须是字符串而必须是表中包含的列名，多个列名 用,分隔。

以上关键字可以自由组合并可以与前面查询语句中的 where，order，limit，skip 等组合使用。

比 如，GameScore 表是游戏玩家的信息和得分表，有 playerName(玩家名称)、score(玩家得分)等你自己创建的列，还有 Bmob 的默认 列 objectId，createdAt，updatedAt,那么我们现在举例如何使用以上的查询关键字来作这个表的统计。

计算总和

计算数据的总和，使用 BmobQuery 提供的 findStatistics 方法，方法原型：

```
virtual void findStatistics(BmobStaticsDelegate* delegate);
```

调用以前先调用 sum 方法将要计算的字段设置好。

如计算 GameScore 表所有玩家的得分总和，sum 后面只能拼接 Number 类型的列名，即要计算哪个列的值的总和，只对 Number 类型有效，多个 Number 列用,分隔，则查询如下：

```
BmobQuery* query = new BmobQuery("GameScore");
query->autorelease();
query->sum("score");
query->findStatistics(this);
```

计算多个列：

```
BmobQuery* query = new BmobQuery("GameScore");
query->autorelease();
query->sum("score,index");
query->findStatistics(this);
```

要监听统计结果，需要实现 BmobStaticsDelegate 接口，必须实现其中的方法：

```
virtual void onStaticsSucess(const void* data){
    //计算成功返回回调
}

virtual void onStaticsError(int code,const void* data){

    //计算失败回调
}
```

分组计算总和

分组计算总和使用 findStatistics 方法，在使用以前先调用 groupby 方法将要分组的字段设置好。

如以 GameScore 创建时间按天统计所有玩家的得分，并按时间降序，groupby 后面只能拼接列名，如果该列是时间类型，则按天分组，其他类型，则按确定值分组：

```
BmobQuery* query = new BmobQuery("GameScore");
query->autorelease();
```

```
query->sum("score");
query->groupby("createdAt");

query->findStatistics(this);
```

多个分组并计算多个列的总和

如以创建时间按天和按玩家名称分组统计所有玩家的得分 1，得分 2 的总和，并按得分 1 的总和降序， groupby 后面只能拼接列名，如果该列是时间类型，则按天分组，其他类型，则按确定值分组：

```
BmobQuery* query = new BmobQuery("GameScore");
query->autorelease();
query->sum("score1,score2");
query->groupby("createdAt,playerName");
query->order("-score1");
query->findStatistics(this);
```

分组计算总和并只返回满足条件的部分值

可以使用 BmobQuery 提供的 where 系列条件实现条件过滤查询，同时要调用 hanving 函数传递 true 参数。如过滤 GameScore 中玩家总分小于 1000 的：

```
BmobQuery* query = new BmobQuery("GameScore");
query->autorelease();
string name = "lingo";
query->sum("score");
query->groupby("playerName");
query->order("-createdAt");
query->having(true);
query->addWhereLessThan("_sumScore",CCInteger::create(1000));

query->findStatistics(this);
```

分组计算总和并返回每个分组的记录数

要查询分组的数量，调用 BmobQuery 的 setHasGroupCount 方法设置需要计数，方法原型是：

```
virtual void setHasGroupCount(bool groupCount) ;
```

比如以创建时间按天统计所有玩家的得分和每一天有多少条玩家的得分记录，并按时间降序：

```
BmobQuery* query = new BmobQuery("GameScore");
query->autorelease();
```

```
query->sum("score");
query->groupby("createedAt");
query->order("-createdAt");
query->setHasGroupCount(true);
query->findStatistics(this);
```

获取不重复的列值

获取不重复的列时，调用 BmobQuery 提供的 groupby 方法设置列名。
如获取玩家分数不重复：

```
BmobQuery* query = new BmobQuery("GameScore");
query->autorelease();
query->groupby("score");
```

其他关键字

average(计算平均值)， max(计算最大值)，min(计算最小值)和 sum 查询语句是类似的。只需要调用对应的方法设置字段即可。

查询 GameScore 表中 score 最小：

```
BmobQuery* query = new BmobQuery("GameScore");

query->autorelease() ;

query->min("score");

query->findStatistics(this);
```

查询 GameScore 表中 score 最大：

```
BmobQuery* query = new BmobQuery("GameScore");

 query->autorelease();

query->max("score");

query->findStatistics(this);
```

查询 GameScore 表中 score 的平均值：

```
BmobQuery* query = new BmobQuery("GameScore");

query->autorelease();

query->average("score");

query->findStatistics(this);
```

用户管理

用户是一个应用程序的核心。对于个人开发者来说，自己的应用程序积累到越多的用户，就会给自己带来越强的创作动力。因此 Bmob 提供了一个专门的用户类——BmobUser 来自动处理用户账户管理所需的功能。

有了这个类，就可以在应用程序中添加用户账户功能。

`BmobUser` 是 `BmobObject` 的一个子类，它继承了 `BmobObject` 所有的方法，具有 `BmobObject` 相同的功能。不同的是，`BmobUser` 增加了一些特定的关于用户账户管理相关的功能。

属性

`BmobUser` 除了从 BmobObject 继承的属性外，还有几个特定的属性：
username: 用户的用户名（必需）。
password: 用户的密码（必需）。
email: 用户的电子邮件地址（可选）。
emailVerified:邮箱认证状态（可选）。
mobilePhoneNumber：手机号码（可选）。
mobilePhoneNumberVerified：手机号码的认证状态（可选）。

扩展用户类

很多时候，你的用户表还会有很多其他字段，如性别、年龄、头像等。那么，你需要对 BmobUser 类进行扩展，添加一些新的属性。示例代码如下所示：

```
class MyUser:public  BmobUser {

public:

    MyUser();

    virtual MyUser();

private:

    CC_SYNTHESIZE(string,m_sex,SEX);

    CC_SYNTHESIZE(int,m_age,Age);

    CC_SYNTHESIZE(string,m_nick,Nick);
```

```
};
```

更多代码实现大家可以下载 SDK，在里面的 `BmobExample` 中查找 `MyUser` 类，参考它的用法。

创建用户对象

创建用户对象如下：

```
BmobUser* user = new BmobUser();

user->autorelease();
```

注册

应用程序可能会要求用户注册。下面的代码是一个典型的注册过程：

```
BmobUser* bu = new BmobUser();

bu→autorelease();

bu->setUsername("sendi");

bu->setPassword("123456");

bu->setEmail("sendi@163.com");

//注意：不能用 save 方法进行注册

bu->enParamsToHttp(“username”,CCString::createWithFormat(“%s”,

                    bu->getUserName().c_str()));

bu->enParamsToHttp(“password”,CCString::createWithFormat(“%s”,

                    bu→getPasswoed().c_str()));

bu->enParamsToHttp(“email”,CCString::createWithFormat(“%s”,
```

```
                    bu→getEmail().c_str()));

bu->signUp(this);

注：注册时必须实现 BmobSaveDelegate 接口，以监听注册结果以及状态，接口中的方法：

    virtual void onSaveSucess(const void* data) {

        //注册成功的回调函数

    }

    virtual void onSaveError(int code,const void* data) {

        //注册失败的回调函数

    }
```

在注册过程中，服务器会对注册用户信息进行检查，以确保注册的用户名和电子邮件地址是独一无二的。此外，对于用户的密码，你可以在应用程序中进行相应的加密处理后提交。

如果注册不成功，可以查看返回的错误对象。最有可能的情况是，用户名或电子邮件已经被另一个用户注册。这种情况你可以提示用户，要求他们尝试使用不同的用户名进行注册。

你也可以要求用户使用 Email 做为用户名注册，这样做的好处是，你在提交信息的时候可以将输入的“用户名“默认设置为用户的 Email 地址，以后在用户忘记密码的情况下可以使用 Bmob 提供重置密码功能。

注：

- 有些时候你可能需要在用户注册时发送一封验证邮件，以确认用户邮箱的真实性。这时，你只需要登录自己的应用管理后台，在应用设置->邮件设置（下图）中把“邮箱验证”功能打开，Bmob 云后端就会在注册时自动发动一封验证给用户。

•username 字段是大小写敏感的字段，如果你希望应用的用户名不区分大小写，请在注册和登录时进行大小写的统一转换。

注：相同的邮箱不能注册不同的账号；同一个账号也不能使用两个邮箱注册。

登录

当用户注册成功后，需要让以后能够用注册的用户名登录到他们的账户使用应用。可以使用 BmobUser 类的 login 方法进行登陆。方法原型：

```
virtual void login(BmobSaveDelegate* delegate);
```

delegate 是登陆监听接口，如果需要获取登陆的状态，就必须实现该接口。

如用户名为 Kiarrier 的登陆：

```
BmobUser* bu = new BmobUser();
```

```
    bu→autorelease();

    bu->setUserName("Kiarrier");

    bu->setPassword("********");

    bu->login(this);
```

也可使用如下方式完成用户名+密码的登录，使用 BmobUser 提供的 loginByAccount 方法登陆，参数为用户名和密码加监听接口，方法原型：

```
    void loginByAccount(string mebileNumber,string pwd,BmobLoginDelegate* delegate);
```

如：

```
    BmobUser* bu = new BmobUser();

    bu→autorelease();

    bu->loginByAccount("15920955603","2222222222222",this);;
```

监听接口 BmobLoginDelegate 的回调方法：

```
virtual void onLoginDone(int code,const void* data){

    //返回登陆状态

}
```

获取当前用户

如果用户在每次打开应用程序时都要登录，这将会直接影响到应用的用户体验。为了避免这种情况，可以使用缓存的 CurrentUser 对象。

每当应用的用户注册成功或是第一次登录成功，都会在本地磁盘中有一个缓存的用户对象，这样，可以通过获取这个缓存的用户对象来进行登录：

```
BmobUser* bmobUser = BmobUser::getCurrentUser();

if(bmobUser != NULL){
```

```
    // 允许用户使用应用

}else{

    //缓存用户对象为空时， 可打开用户注册界面...

}
```

在扩展了用户类的情况下获取当前登录用户，可以使用如下的示例代码（`MyUser`类可参看上面）：

```
MyUser* userInfo = BmobUser::getCurrentUser();
```

退出登录

退出登录非常简单，可以使用如下的代码：

```
BmobUser::logOut();   //清除缓存用户对象

BmobUser* currentUser = BmobUser::getCurrentUser(); // 现在的 currentUser 是 NULL 了
```

更新用户

用户可能需要修改信息，应用具备修改个人资料的功能，修改个人资料使用 BmobUser 提供的 update 方法更新信息，Bmob 提供的用户更新方式有两种：

第一种：`新建一个用户对象，并调用 update(string objectId,BmobSaveDeleaget* delegate)方法来更新（推荐使用）`，示例：

```
BmobUser* newUser = new BmobUser();

newUser→autorelease();

newUser->setEmail("xxx@163.com");
```

```
BmobUser bmobUser* = BmobUser::getCurrentUser();

newUser->update(bmobUser->getObjectId(),this)
```

第二种：获取本地的用户对象，并调用 update（BmobUpdateDelegate* delegate）方法来更新（不推荐使用），示例：

```
BmobUser* bmobUser = BmobUser::getCurrentUser();

if(bmobUser != NULL){

    // 修改用户的邮箱为 xxx@163.com

    bmobUser->setEmail("xxx@163.com");

    bmobUser->update(this);

}
```

上面的两种更新方法都如果需要监听修改状态，都需要传递一个监听对象指镇给 update 方法，该对象必须实现 BmobUpdateDelegate 接口，其中必须实现两个回调方法：

```
    virtual void onUpdateSucess(const void* data){

        //用户信息更新成功的回调

    }

    virtual void onUpdateError(int code,const void* data) {

        //用户信息更新失败的回调

    }
```

1、开发者在进行用户更新操作的时候，推荐使用第一种方式来进行用户的更新操作,因为此方法只会更新你提交的用户信息（比如只会向服务器提交当前用户的 email 值），而不会将本地存储的用户信息也提交到后台更新。

2、在更新用户信息时，如果用户邮箱有变更并且在管理后台打开了邮箱验证选项的话，Bmob 云后端同样会自动发一封邮件验证信息给用户。

查询用户

查询用户和查询普通对象一样，使用 BmobQuery 中的 findObjects 方法并传递一个实现的监听接口参数查询，方法原型：

```
virtual void findObjects(BmobFindDelegate* delegate) = 0;
```

如下：

```
BmobQuery* query = new BmobQuery(BmobSDKInit::USER_TABLE);

query→autorelease();

string name = "lingo";

query->addWhereEqualTo("username",CCString::createWithFormat("%s",

                                name.c_str()));

query->findObjects(this);
```

注：要查询用户信息，在创建 BmobQuery 对象时，必须传递 BmobSDKInit::USER_TABLE 作为参数。

浏览器中查看用户表

User 表是一个特殊的表，专门存储 BmobUser 对象。在浏览器端，你会看到一个 User 表旁边有一个小人的图标。

密码重置

有了密码系统，肯定会有用户忘记密码的情况。对于这种情况，我们提供了以下两种方法，让用户安全地重置密码。

邮箱重置密码

使用邮箱重置密码，使用 BmobUser 提供的 resetPasswordByEamil 方法并传递邮箱地址和监听接口指针，开发者只需要求用户输入注册时的电子邮件地址即可：

```
BmobUser* bu = new BmobUser();

bu->autorelease();

 bu->resetPasswordByEmail("xxxx@bmob.com",this);
```

需要监听重置的状态，开发者必须实现 BmobResetPasswordDelegate 接口，上面传递给 resetPasswordByEmail 的第二个参数就是实现该接口的指针，接口中的回调方法：

```
    virtual void onResetSucess(const void* data){

        //传送重置密码的邮件成功

    }

    virtual void onResetError(int code,const void* data){

        //重置错误

    }
```

邮箱重置密码的流程如下：

1.用户输入他们的电子邮件，请求重置自己的密码。

2.Bmob 向他们的邮箱发送一封包含特殊的密码重置链接的电子邮件。

3.用户根据向导点击重置密码连接，打开一个特殊的 Bmob 页面，根据提示他们可以输入一个新的密码。

4.用户的密码已被重置为新输入的密码。

手机号码重置密码

Bmob 引入了短信验证系统，如果用户已经验证过手机号码或者使用过手机号码注册或登录

过，可以通过手机号码来重置用户密码，以下是官方建议使用的重置流程：

1、请求手机重置密码必须先获取短信验证码，获取短信验证码使用 BmobUser 提供的 requestSMSCode 方法，传入电话号码和模板名以及请求监听接口，方法原型是：

```
void requestSMSCode(string meblieNumber,string template_name,
                    BmobRequestSMSCodeDelegate* delegate);
```

注：如果在后台设置模板名，在申请验证码时需要传入 template_name。

```
BmobUser* bu = new BmobUser();

bu→autorelease();

 bu->requestSMSCode("159.....","",this);
```

监听请求状态，需要实现 BmobRequestSMSCodeDelegate 接口，上面的 this 实现了该接口，接口中的回调函数：

```
virtual void onRequestDone(int code,const void* data){
    //返回请求状态
}
```

2、用户收到重置密码的验证码之后，就可以调用 resetPasswordBySMSCode 方法来实现密码重置，其中传递重置的密码和短信验证码以及监听接口，方法原型是:

```
void resetPasswordBySMSCode(string pw,string code,
                    BmobResetPasswordByCodeDelegate* delegate);
```

如：

```
BmobUser* bu = new BmobUser();

bu→autorelease();

bu->resetPasswordBySMSCode(psw,msm_code,this);
```

需要监听重置状态，需要实现 BmobResetPasswordByCodeDelegate 接口，回调函数：

```
virtual void onResetDone(int code,const void* data) {
    //返回重置状态
```

```
}
```

重置成功以后，用户就可以使用新密码登陆了。

注：

1、请开发者按照官方推荐的操作流程来完成重置密码操作。也就是说，开发者在进行重置密码操作时，无需调用 verifySmsCode 接口去验证该验证码的有效性。

2、验证码只能使用一次，一旦该验证码被使用就会失效，那么再拿失效的验证码去调用重置密码接口，一定会报 207-验证码错误。因为重置密码接口已经包含验证码的有效性验证。

密码修改

SDK 为开发者提供了直接修改当前用户登录密码的方法，只需要传入旧密码和新密码，然后调用 BmobUser 提供的方法 updateCurrentUserPassword 即可，方法原型：

```
void updateCurrentUserPassword(string old_pwd,string new_pwd,
                               BmobUpdateDelegate* delegate);
```

以下是示例：

```
BmobUser* bu = new BmobUser();

bu→autorelease();

bu->updateCurrentUserPassword(“旧密码”，”新密码”,this);
```

需要监听该方法的修改状态，需要实现 BmobUpdateDelegate 监听接口。

修改成功的返回是 JSON 数据，如：

```
{
    "msg":"ok"
}
```

注：此方法修改密码时，同样需要使用用户的 ID 以及为了安全 **X-Bmob-Session-Token**，**所以需要用户登陆才能修改密码。**

邮箱验证

设置邮件验证是一个可选的应用设置，这样可以对已经确认过邮件的用户提供一部分保留的体验，邮件验证功能会在用户(User)对象中加入 emailVerified 字段，当一个用户的邮件被新添加或者修改过的话，emailVerified 会被默认设为 false，如果应用设置中开启了邮箱认证功能，Bmob 会对用户填写的邮箱发送一个链接，这个链接可以把 emailVerified 设置为 true.

emailVerified 字段有 3 种状态可以考虑：

- •true：用户可以点击邮件中的链接通过 Bmob 来验证地址，一个用户永远不会在新创建这个值的时候显示 emailVerified 为 true。
- •false：用户(User)对象最后一次被刷新的时候，用户并没有确认过他的邮箱地址，如果你看到 emailVerified 为 false 的话，你可以考虑刷新用户(User)对象。
- •missing：用户(User)对象已经被创建，但应用设置并没有开启邮件验证功能； 或者用户(User)对象没有 email 邮箱。

请求验证 Email

发送给用户的邮箱验证邮件会在一周内失效，可以通过调用 `requestEmailVerify` 来强制重新发送，方法原型：

```
void requestEmailVerify(string email,BmobEmailVerifyDelegate* delegate);
```

如：

```
BmobUser* bu = new BmobUser();

bu→autorelease();

bu->requestEmailVerify("914143799@qq.com",this);
```

监听验证状态必须实现 BmobEmailVerifyDelegate 接口，接口中回调函数：

```
virtual void onEmailVerifySucess(const void* data) {
    // "请求验证邮件成功，请到" + email + "邮箱中进行激活。"//
}
virtual void onEmailVerifyError(int code,const void* data){
    //请求验证邮件失败
}
```

手机号码验证

请求发送短信验证码

Bmob 自 V3.3.9 版本开始引入了短信验证系统，可通过 requestSMSCode 方式请求发送短信验证码：

```
BmobUser* bu = new BmobUser();

bu→autorelease();

 bu->requestSMSCode("159.....","",this);
```

监听请求状态，需要实现 BmobRequestSMSCodeDelegate 接口，上面的 this 实现了该接口，接口中的回调函数：

```
    virtual void onRequestDone(int code,const void* data){

        //返回请求状态

    }
```

短信默认模板：

```
    您的验证码是`%smscode%`，有效期为`%ttl%`分钟。您正在使用`%appname%`的验证码。【比目科技】
```

注：

1、模板名称：模板名称需要开发者在应用的管理后台进行短信模板的添加工作，具体：短信服务->短信模板,之后点击创建即可。

具体请看下图：

2、只有审核通过之后的自定义短信模板才可以被使用，如果自定义的短信模板其状态显示审核中或者审核失败,再调用该方法则会以默认模板来发送验证码。

3、开发者提交短信验证码模板时需注意以下几点：

1）、模板中不能有【】和 []，否则审核不通过；

2）、如果你提交的短信模板无法发送，则有可能包含一些敏感监控词，具体可去 Github 下载 短信关键字监控参考文档 **来查看提交内容是否合法。**

3）、一天一个应用给同一手机号发送的短信不能超过 10 条，否则会报 10010 错误，其他错误码可查看 短信功能相关错误码 **。**

邮箱登录

新增邮箱+密码登录方式,可以通过 loginByAccount 方法来操作，使用方法查看账号名加密码登陆：

```
virtual void loginByAccount(account, password, this);
```

手机号码登录

在手机号码被验证后，用户可以使用该手机号码进行登录操作，使用方法查看账号名加密码登陆。

手机号码登录包括两种方式：手机号码＋密码、手机号码＋短信验证码。

手机号码+密码

```
BmobUser.loginByAccount( "11 位手机号码", "用户密码", this);
```

手机号码+短信验证码

先请求登录的短信验证码：

```
BmobUser* bu = new BmobUser();
```

```
bu->requestSMSCode( "11 位手机号码","模板名称",this);
```

注：请求验证码查看

最后调用 loginBySMSCode 方法进行手机号码登录:

```
BmobUser* bu = new BmobUser();

bu->autorelease();

bu->loginBySMSCode("15920955603","160469",this);
```

手机号码一键注册或登录

Bmob 同样支持手机号码一键注册或登录，以下是一键登录的流程：

1、请求登录操作的短信验证码：

```
BmobSMS.requestSMSCode(context, "11 位手机号码","模板名称", this);
```

2、用户收到短信验证码之后，就可以调用 signOrLoginByMobilePhone 方法来实现一键登录，方法原型是：

```
void signOrLoginByMobilePhone(string mebileNumber,string code,
                              BmobLoginDelegate* delegate);
```

如:

```
BmobUser.signOrLoginByMobilePhone(this, "11 位手机号码", "验证码", this);
```

可以查看验证码获取和手机+验证码登陆。

绑定手机号码

如果已有用户系统，需要为用户绑定手机号，那么官方推荐的绑定流程如下：

第一步、先发送短信验证码并验证验证码的有效性,即调用 requestSMSCode 发送短信验证码，调用 verifySmsCode 来验证有效性。

第二步、在验证成功之后更新当前用户的 MobilePhoneNumber 和

MobilePhoneNumberVerified 两个字段，具体绑定示例如下：

```
/**

* bind mobile

*/

BmobUser* bu = new BmobUser();

bu->setMobilePhoneNumber("15920955603");

bu->setMobilePhoneNumberVerified(true);

bu→clear();

 bu->enParamsToHttp("mobilePhoneNumber",CCString::createWithFormat("%s",bu->getMobilePhoneNumber().c_str()));

bu->enParamsToHttp("mobilePhoneNumberVerified",CCBool::create(bu->

                                CCBool::create(bu->getMobilePhoneNumberVerified()));

BmobUser* cur = BmobUser::getCurrentUser();

bu->update(cur->getObjectId(),this);
```

可以查看更新用户部分。